A calculus of Gamma programs

Chris Hankin, Daniel Le Métayer, David Sands

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1992

5ème anniversaire

N° 1758

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

A CALCULUS OF GAMMA PROGRAMS

Chris HANKIN
Daniel LE METAYER
David SANDS

Octobre 1992

RR-1758

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

A Calculus of Gamma programs

Chris Hankin[1] , Daniel Le Métayer[2] and David Sands[3]
Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate
LONDON SW7 2BZ

July 1992
Publication Interne n°674 - Juillet 1992 - 32 pages - Programme 1

Abstract

Gamma is a minimal language based on conditional multiset rewriting. The virtues of this paradigm in terms of systematic program construction and design of programs for highly parallel machines have been demonstrated in previous papers. We introduce here sequential and parallel operators for combining Gamma programs and we study their properties. The main focus of the paper is to give conditions under which sequential composition can be transformed into parallel composition and vice versa. Such transformations are especially valuable for adapting Gamma programs for execution on a particular target architecture.

Une analyse des programmes Gamma

Résumé

Gamma est un langage minimal reposant sur la récriture conditionnelle de multi-ensembles. Les qualités de ce formalisme pour la construction systématique de programmes ont déja été démontrées par ailleurs. Nous introduisons ici un opérateur séquentiel et un opérateur parallèle pour la composition de programmes Gamma et nous étudions leurs propriétés. Nous portons une attention particulière aux conditions d'équivalence entre composition parallèle et composition séquentielle. Ces conditions sont utiles pour adapter les programmes Gamma à des architectures particulières.

[1] Partially funded by ESPRIT BRAs 3074 (SemaGraph) and 6809 (Semantique).

[2] On leave from IRISA, Campus de Beaulieu, Rennes, France and partially funded by SERC Visiting Fellowship GR/ H 19330.

[3] Partially funded by ESPRIT BRA 6809 (Semantique).

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE (URA 227) UNIVERSITE DE RENNES I INSA DE RENNES
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE (UNITE DE RECHERCHE DE RENNES)

A Calculus of Gamma programs

Chris Hankin, Daniel Le Métayer and David Sands
Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate
LONDON SW7 2BZ

July 1992

1. Introduction

We have seen in the last decade the emergence of a new generation of parallel machines featuring a huge number of processors, the most significant representative being the Connection Machine [9] with its 64k processing elements. In order to make these architectures upgradable and cost effective, individual components are kept as simple as possible and they are connected in a loosely coupled way. These unique features impose the need for the design of new programming techniques for exploiting these machines efficiently. It is very unlikely that a programmer can mentally manage the details of the decomposition of a program into a great number of individual tasks. The programmer should rather be provided with a high-level programming language in which parallelism is left implicit and an associated compiler which is able to detect this implicit parallelism and to map it effectively onto the parallel architecture. So these massively parallel machines raise interesting challenges in the fields of parallel language design and compiler construction.

The Gamma formalism was proposed a few years ago to allow the description of programs without artificial sequentiality (by artificial we mean sequentiality that is not implied by the logic of the program). A simple example illustrates this point. The problem is to find the maximum element of a non-empty set. In an imperative language the set can be represented as an array a[1:n] and a possible program is:

```
maxset1:    m:= a[1];
            i := 1;
            * [i < n  →
                  i := i+1;
                  m:= max(m, a[i])]
```

While the condition i < n holds, index i is incremented and a new value of m is computed.
In a functional language, the set would be represented as a list and the program would be:

maxset2(l) = **if** tail(l) = nil **then** head(l) **else** max (head(l),maxset2(tail(l)))

In both cases the program imposes a total ordering on the comparisons of the elements: the first element is compared with the second, then their maximum is compared with the third, and so forth. In both formalisms one could imagine a less constraining solution involving explicit or implicit parallelism. For example, in the functional language a divide-and-conquer version of the above program would be:

maxset3(l) = **if** tail(l) = nil **then** head(l) **else**
let (l1,l2) = split(l) **in**
max(maxset3(l1), maxset3(l2))

where split (l1,..., ln) = ((l1,...,ln/2), (ln/2+1,...,ln)). Here again a (partial) ordering is imposed on the comparisons: for example the first and last elements of the list will not be compared (except in the case where they are the maxima of their respective sublists).
In fact the maximum of a set can be computed by performing the comparisons of the elements in any order. So we would like an abstract algorithm of the form:

while there are at least two elements in the set
select two elements of the set, **compare** them and **remove** the smaller one

This is almost a Gamma program. In Gamma such a statement can be written as follows:

maxset4: x, y $\rightarrow$ y $\Leftarrow$ x $\leq$ y

x $\leq$ y specifies a property to be satisfied by the selected elements x and y; these elements are replaced in the set by the value y. Nothing is said in this definition about the order of evaluation of the comparisons; if several disjoint pairs of elements satisfy x $\leq$ y, the comparisons and replacements can even be performed in parallel. An intuitive way of describing the meaning of a Gamma program is the metaphor of the chemical reaction: the set can be seen as a chemical solution; R(x,y) = x $\leq$ y is called the reaction condition and A(x,y) = {y} is the action. The computation terminates when a stable state is reached, that is to say when no elements of the set satisfy the reaction condition.

We illustrate the programming style of the language with a few examples. The following program returns the set of prime numbers smaller than n.

pn(n) = prime_numbers ({2, ..., n})
prime_numbers: x, y $\rightarrow$ y $\Leftarrow$ multiple (x,y)

where multiple(x,y) holds if x is a multiple of y. The program proceeds by removing from the set {2, ..., n} elements that are multiples of another element in the set. Next we consider the sorting problem: the goal is to organize the elements of an array in increasing order. We use a multiset of pairs (index,value) and the program exchanges ill-ordered values until all values are well-ordered.

sort: (i,v), (j,w) $\rightarrow$ (i,w), (j,v) $\Leftarrow$ (i > j) and (v < w)

The majority element of a multiset M is an element occurring more than card(M)/2 times in the multiset. We propose a solution to the problem of finding the majority element, assuming that such an element exists:

maj_elem(M) = m where
{m} = $P_2(P_1(M))$
P_1: x, y $\rightarrow$ $\Leftarrow$ x $\neq$ y
P_2: x, y $\rightarrow$ x $\Leftarrow$ True

The interested reader may find in [4] a longer series of examples (string processing problems, graph problems, geometric problems, ...) illustrating the Gamma style of programming. The possibility of getting rid of artificial sequentiality in Gamma has two important consequences:

- It makes Gamma suitable as an intermediate language in the program derivation process allowing the programmer to design a very abstract version of his program in the first place (which is easier to prove correct); this version is then specialized for the sake of efficiency by introducing extra sequentiality. The benefit of using Gamma in systematic program construction is illustrated in [3,7,14].
- Gamma programs do not have any sequential bias and the language can be implemented very naturally on parallel machines. The interested reader may refer to [2,6] for the description of parallel implementations of Gamma.

So far we have only (and implicitly) mentioned one possible way of combining Gamma programs: the "functional" sequential composition that occurs in the definition of maj_elem. For the sake of modularity it is desirable that a language offers a rich set of operators for combining programs. It is also fundamental that these operators enjoy a useful collection of algebraic laws in order to

make it possible to reason about composed programs. This paper is mainly devoted to the presentation of two operators for the combination of Gamma programs, namely the sequential composition $P_1 \circ P_2$ and the parallel composition $P_1 + P_2$ and to the study of their properties. The intuition behind $P_1 \circ P_2$ is that the stable multiset reached after the execution of P_2 is given as argument to P_1. On the other hand the result of $P_1 + P_2$ is obtained by executing the reactions of P_1 and P_2 in parallel. The paper defines conditions under which $P_1 \circ P_2$ can be transformed into $P_1 + P_2$ and vice-versa. These transformations are useful to improve the efficiency of a program on a particular machine. Let us take the following example [4] to illustrate this point:

connected = singleton $\circ$ ($P_1 + P_2$) where

P_1: v, w, (m,n) $\rightarrow$ v + w $\Leftarrow$ vertices(v) and vertices(w) and m $\in$ v and n $\in$ w

P_2: v, (m,n) $\rightarrow$ v $\Leftarrow$ vertices(v) and m $\in$ v and n $\in$ v

This program is used to detect whether a graph is connected or not. The program is applied to a multiset representation of a graph; vertices are represented by singleton sets containing the vertex and edges are represented by pairs of vertices. The program proceeds by building bigger and bigger aggregates of connected vertices (through P_1). The predicate *vertices* is true when the multiset element supplied as an argument is a set of vertices. P_2 is used to remove from the multiset edges connecting two vertices belonging to the same set (of connected vertices). The graph is connected if all the vertices can be gathered into a single set, which is tested via the primitive *singleton*. Our algebra of programs allows us to show that:

$$P_1 + P_2 = P_2 \circ P_1$$

which means that all the reactions of P_2 can be postponed until no more P_1 reaction can take place. If the target architecture is a sequential one (or even a parallel one with relatively few processors) $P_2 \circ P_1$ will be more efficient because many useless tests of the reaction condition of P_2 will be avoided; however $P_1 + P_2$ might turn out to be a better version if executed on a massively parallel machine because unnecessary edges can be removed by P_2 at the same time as aggregates are built by P_1.

The next section gives a formal definition of the syntax and the semantics of Gamma programs and their composition operators. Section 3 is devoted to the Gamma program calculus. What makes reasoning about Gamma programs more difficult is the fact that they may be nondeterministic in general. However most useful programs written in Gamma turn out to be deterministic and we give in section 4 sufficient conditions to ensure this property. This allows us to decompose the proof of $P_1 = P_2$ into two parts: ($P_1 \leq P_2$) and (deterministic (P_2)), $P_1 \leq P_2$ meaning that P_2 can produce all the results that P_1 can return. Previous experience has shown that a small number of paradigms are systematically used to write Gamma programs. These paradigms are presented in section 5 and we apply the theory established in previous sections to show that

they enjoy useful properties. When the transformation rules described in section 3 do not apply, it may still be possible to replace sequential composition by parallel composition using a technique called pipelining. This transformation is presented in section 6. In conclusion we discuss a way of enriching our calculus and we identify avenues for further research.

2. Syntax and Semantics of Gamma Programs

The syntax of Gamma programs is defined as follows:

$P \in$ Programs

$R \in$ Reactions

$A \in$ Actions

$P \quad ::= \quad (R,A) \mid P_1 \circ P_2 \mid P_1 + P_2$

We do not specify the form of reactions and actions; in any (R,A) pair, the reaction and the action are functions of the same arity and the reaction is a predicate whereas the action produces a tuple of multiset elements. We will often use the alternative syntax for basic programs (as was already done in the introduction):

$G: x_1, \ldots, x_n \rightarrow A(x_1,\ldots,x_n) \Leftarrow R(x_1,\ldots,x_n)$

for:

$G:(R,A)$

where R and A are of arity n.

The semantics of Gamma programs is given in the style of Plotkin's Structural Operational Semantics. Configurations consist of a program and a multiset; transitions are of the form:

$(P,M) \rightarrow (P',M')$

or $(P,M) \rightarrow M'$

Multisets are terminal configurations. The semantics are presented in the figure.

To prove certain properties of programs we need to establish properties of intermediate forms. First we concentrate on the intermediate states of the programs via a reduction ordering on programs.

Definition 1: The "reduction" ordering on programs, $\rightsquigarrow$, is defined as the least relation satisfying:

- $$\frac{Q \rightsquigarrow Q'}{P \circ Q \rightsquigarrow P \circ Q'}$$
- $$\frac{\neg(Q \rightsquigarrow)}{P \circ Q \rightsquigarrow P}$$

- $\dfrac{P \rightsquigarrow P'}{P + Q \rightsquigarrow P' + Q}$
- $\dfrac{Q \rightsquigarrow Q'}{P + Q \rightsquigarrow P + Q'}$

[]

$((R,A),M) \rightarrow ((R,A),(M - \{x1,\ldots xn\} \cup A(x1,\ldots xn)))$ if $x1,\ldots xn \in M \wedge R(x1,\ldots xn)$

$((R,A),M) \rightarrow M$ if $\neg\exists x1,\ldots xn \in M.R(x1,\ldots xn)$

$$\frac{(P2,M) \rightarrow M}{(P1 \circ P2,M) \rightarrow (P1,M)} \qquad \frac{(P2,M) \rightarrow (P2', M')}{(P1 \circ P2,M) \rightarrow (P1 \circ P2',M')}$$

$$\frac{(P1,M) \rightarrow M \qquad (P2,M) \rightarrow M}{(P1 + P2,M) \rightarrow M} \qquad \frac{(P1,M) \rightarrow (P1',M')}{(P1 + P2,M) \rightarrow (P1' + P2,M')}$$

and similarly for P2

Semantics of Gamma

Where $\neg(Q \rightsquigarrow)$ is shorthand for $\neg\exists Q'.Q \rightsquigarrow Q'$. Let $\rightsquigarrow^*$ denote the transitive reflexive closure of $\rightsquigarrow$. Inspection of the rules shows that the program is transformed as it is reduced; in particular, sequential compositions are deleted. This observation leads us to make the following definition:

Definition 2: We define the *residual* part of a program P, written $\underline{\underline{P}}$, by induction on the syntax of P:

$$\underline{\underline{(R,A)}} = (R,A)$$
$$\underline{\underline{P_1 \circ P_2}} = \underline{\underline{P_1}}$$
$$\underline{\underline{P_1 + P_2}} = \underline{\underline{P_1}} + \underline{\underline{P_2}}$$

[]

Definition 3: P is *simple* if $P = \underline{\underline{P}}$.

The intuition is that P is simple if it does not contain sequential composition.

Proposition 1.

(i) $\rightsquigarrow^*$ is confluent and terminating

(ii) ($P \rightsquigarrow^* Q$ and $\neg(Q \rightsquigarrow)$) $\Leftrightarrow Q = \underline{P}$

[]

(i) says that if $P \rightsquigarrow^* Q$ and $P \rightsquigarrow^* R$ then there is an S such that $Q \rightsquigarrow^* S$ and $R \rightsquigarrow^* S$ and that there are no infinite $\rightsquigarrow$-reduction sequences. (ii) says that the $\rightsquigarrow$-normal forms are characterised by $\underline{=}$.

Proposition 2.

$(P,M) \rightarrow M \Rightarrow P$ is simple

[]

Definition 4: The *active* part of a program P, written $\{P\}$, is defined by induction on the syntax of P:

$\{(R,A)\} = (R,A)$

$\{P \circ Q\} = \{Q\}$

$\{P + Q\} = \{P\} + \{Q\}$

[]

Notice that the active part of a program is a (kind of) dual to the residual part; it characterises the reactions that are "immediately" applicable.

The next theorem states that each transition in the operational semantics of Gamma is either a reduction of the program or a modification of the multiset; in either case, the other component remains unchanged.

Theorem 1.

If $(P_1,M) \rightarrow (P_2,M')$ then:

either (i) $P_1 \rightsquigarrow P_2$ and $M = M'$

or (ii) $P_1 = P_2$ and $(\{P_1\},M) \rightarrow (\{P_1\},M')$

Proof (By structural induction on P_1)

We just consider the case for $P_1 = P \circ Q$. There are two cases according to the last inference that $(P \circ Q,M) \rightarrow (P_2,M')$:

(1). $(Q,M) \rightarrow M$ and hence $(P \circ Q,M) \rightarrow (P,M)$. But then by Proposition 2, Q is simple. By Proposition 1(ii) $\neg(Q \rightsquigarrow)$ and so $P \circ Q \rightsquigarrow P$ as required.

(2) $(Q,M) \rightarrow (Q',M')$ and $(P \circ Q,M) \rightarrow (P \circ Q',M')$. By the induction hypothesis, we have two

subcases:

(a) $Q = Q'$ and $(\{Q\},M) \rightarrow (\{Q\},M')$, in which case $(\{P \circ Q\},M) \rightarrow (\{P \circ Q\},M')$ by definition of active parts.

(b) $Q \rightsquigarrow Q'$ and $M = M'$, in which case $P \circ Q \rightsquigarrow P \circ Q'$ by definition of $\rightsquigarrow$.

[]

Then we have the following result which intuitively says that all of the work is done by the active part of the program and any work that the active part is capable of is done .

Theorem 2.

$(P,M) \rightarrow (P,M') \Leftrightarrow (\{P\},M) \rightarrow (\{P\},M')$

Proof

($\Rightarrow$) immediate from Theorem 1.

($\Leftarrow$) simple induction on the structure of P.

[]

These two previous results establish the role of active parts of the program in transition sequences; we now turn to the role of residual parts - the following theorem establishes an important factorisation result for transition sequences:

Theorem 3.

$(P,M) \rightarrow^* N \Leftrightarrow (P,M) \rightarrow^* (\underline{\underline{P}},N) \rightarrow N$

Proof

($\Leftarrow$) immediate.

($\Rightarrow$) We have $(P,M) \rightarrow^* (Q,N) \rightarrow N$ for some Q. By Proposition 2, Q is simple. By Theorem 1, $P \rightsquigarrow^* Q$. Since Q is simple, $\neg(Q \rightsquigarrow)$, so by Proposition 1, $Q = \underline{\underline{P}}$.

[]

We close this section by introducing the notion of active contexts and establishing some of their properties which will be useful in the following.

Definition 5. An *active context*, **A**, is a term containing a single hole in its active part:

$$\mathbf{A} \quad ::= \quad [] \mid P + \mathbf{A} \mid \mathbf{A} + P \mid P \circ \mathbf{A}$$

A[P] denotes active context **A** with program P in place of the hole.

[]

Theorem 4.

$\forall A.(P,M) \rightarrow (P',M') \Rightarrow (A[P],M) \rightarrow (A[P'],M')$

Proof

Easy induction on the structure of active contexts.

[]

Corollary 1.

$\forall A.(P,M) \rightarrow^* (Q,N) \Rightarrow (A[P],M) \rightarrow^* (A[Q],N)$

[]

3. The Gamma Calculus

We start this section with some technical results concerning $\circ$ and + and their relationship.

Definition 6: We define the following divergence relation, "may diverge" written $\uparrow$, on program configurations:

$$(P,M)\uparrow \equiv \exists\{(P_i,M_i) \mid i \in \omega\}.\ (P_0,M_0) = (P,M) \text{ and } (P_i,M_i) \rightarrow (P_{i+1},M_{i+1})$$

[]

The following proposition shows that $\circ$ indeed behaves as a composition.

Proposition 3.

$$(\exists N'.\ (P_2,M) \rightarrow^* N' \ \&\ (P_1,N') \rightarrow^* N) \Leftrightarrow (P_1 \circ P_2,M) \rightarrow^* N$$

Proof

($\Rightarrow$) By Theorem 3, $(P_2,M) \rightarrow^* N'$ implies that $(P_2,M) \rightarrow^* (\underline{\underline{P}}_2,N')$ and $(\underline{\underline{P}}_2,N') \rightarrow N'$. Then by Corollary 1, we have $(P_1 \circ P_2,M) \rightarrow^* (P_1 \circ \underline{\underline{P}}_2,N')$ and thus, using the last step, $(P_1 \circ P_2,M) \rightarrow^* (P_1,N')$ and the result follows using the second premise.

($\Leftarrow$) We prove the slightly stronger result that:

$$(P_1 \circ P_2,M) \rightarrow^k N \Rightarrow \exists N'.(P_2,M) \rightarrow^{k1} N' \text{ and } (P_1,N') \rightarrow^{k2} N$$

with $k = k1 + k2$. The proof is a straightforward induction over the length of the derivation, the basis being vacuous.

[]

Proposition 4.

$$(P_1 \circ P_2,M) \rightarrow^* N \Rightarrow (P_1 + P_2,M) \rightarrow^* (\underline{\underline{P}}_1 + \underline{\underline{P}}_2,N)$$

Proof

By Proposition 3, $\exists N'.(P_2,M) \rightarrow^* N'$ and $(P_1,N') \rightarrow^* N$. Then by Theorem 3, $(P_2,M) \rightarrow^*$

$(\underline{\underline{P}}_2,N')$ and $(P_1,N') \rightarrow^* (\underline{\underline{P}}_1,N)$. Applying Corollary 1 twice gives:

$$(P_1 + P_2,M) \rightarrow^* (P_1 + \underline{\underline{P}}_2,N') \rightarrow^* (\underline{\underline{P}}_1 + \underline{\underline{P}}_2,N)$$

[]

Proposition 5.

$$(P \circ Q,M)\uparrow \Rightarrow (P + Q,M)\uparrow$$

Proof

We split $(P \circ Q,M)\uparrow$ into two possible cases:

(i) $\exists\{(Q_i,M_i) \mid i \in \omega\}.\ (Q,M) \rightarrow (Q_1,M_1)\ \&\ (Q_i,M_i) \rightarrow (Q_{i+1},M_{i+1})$

Then we have $(P + Q,M) \rightarrow (P + Q_1,M_1)$ and $(P + Q_i,M_i) \rightarrow (P + Q_{i+1},M_{i+1})$ and hence, $(P+Q,M)\uparrow$.

(ii) $(P \circ Q,M) \rightarrow^* (P,M')$ and $(P,M')\uparrow$

As in Proposition 4, we reason that:

$(P + Q,M) \rightarrow^* (P + \underline{\underline{Q}},M')$ and $(P,M')\uparrow$ implies that $(P + \underline{\underline{Q}},M')\uparrow$

[]

Definition 7: We define the predicate Φ : Prog × Multi →Bool by:

$$\Phi(P,M) \Leftrightarrow (\underline{\underline{P}},M) \rightarrow M$$

[]

Φ is a post-condition for programs, viz:

Proposition 6.

$$(P,M) \rightarrow^* N \Rightarrow \Phi(P,N)$$

Proof

By Theorem 3.

[]

Proposition 7.

$$\Phi(P,M) \text{ and } N \text{ is a submultiset of } M \Rightarrow \Phi(P,N)$$

This property is a consequence of the locality of reaction conditions.

In order to establish further properties of the two combinators, we introduce an ordering between programs which captures the notion of "refinement". First we define the capability function of a program:

Definition 8: We define the capability function, $\mathbb{C}$: Prog → Multi → $\wp(\text{Multi}_\perp)$ where Multi is the set of finite multisets and $\_\perp$ is the usual lifting operator, thus:

$$\mathbb{C}(P)M = \{\bot \mid (P,M)\uparrow\} \cup \{N \mid (P,M) \rightarrow^* N\}$$

[]

We can construct a number of orderings on programs, by considering the ordering pointwise on $\mathbb{C}(P)$.

(a)Relational Ordering: For $S_1, S_2 \in \wp(\text{Multi}_\bot)$:

$$S_1 \leq_R S_2 \Leftrightarrow S_1 \text{ is a subset of } S_2$$

The ordering on programs, is defined:

$$P_1 \leq_R P_2 \Leftrightarrow \forall M.\ \mathbb{C}(P_1)\ M \leq_R \mathbb{C}(P_2)\ M$$

which may be written more explicitly:

$$P_1 \leq_R P_2 \Leftrightarrow \forall M.\ ((P_1,M)\uparrow \Rightarrow (P_2,M)\uparrow)\ \&\ (\forall N.(P_1,M) \rightarrow^* N \Rightarrow (P_2,M) \rightarrow^* N)$$

The other orderings use the ordering $\leq$ on the flat domain $\text{Multi}_\bot$.

(b) The lower (Hoare) ordering: For $S_1, S_2 \in \wp(\text{Multi}_\bot)$:

$$S_1 \leq_L S_2 \Leftrightarrow \forall L_1 \in S_1.\ \exists L_2 \in S_2.\ L_1 \leq L_2$$

The corresponding ordering on programs is:

$$P_1 \leq_L P_2 \Leftrightarrow \forall M.\ \forall N.\ (P_1,M) \rightarrow^* N \Rightarrow (P_2,M) \rightarrow^* N$$

(c) The upper (Smyth) ordering: For $S_1, S_2 \in \wp(\text{Multi}_\bot)$:

$$S_1 \leq_U S_2 \Leftrightarrow \forall L_2 \in S_2.\ \exists L_1 \in S_1.\ L_1 \leq L_2$$

The corresponding ordering on programs is:

$$P_1 \leq_U P_2 \Leftrightarrow \forall M.((P_2,M)\uparrow \Rightarrow (P_1,M)\uparrow)\ \&$$
$$((P_1,M)\uparrow \text{ or } (\forall N.(P_2,M) \rightarrow^* N \Rightarrow (P_1,M) \rightarrow^* N)))$$

In the following, we write $(P,M)\downarrow^{must}$ if $\bot \notin \mathbb{C}(P)M$ and $P\downarrow^{must}$ if $\forall M.(P,M)\downarrow^{must}$. The ordering (c) may be equivalently written:

$$P_1 \leq_U P_2 \Leftrightarrow \forall M.(P_1,M)\downarrow^{must} \Rightarrow ((P_2,M)\downarrow^{must}\ \&$$
$$(\forall N.(P_2,M) \rightarrow^* N \Rightarrow (P_1,M) \rightarrow^* N))$$

Notice that $P_1 \leq_R P_2$ may be read as "P_1 correctly implements P_2" - if the programmer is willing to accept any result produced by P_2, then any result from P_1 must be acceptable.

It is straightforward to demonstrate the following simple relationships between orderings:

- $P_1 \leq_R P_2 \Rightarrow P_1 \leq_L P_2\ \&\ P_1 \geq_U P_2$
- $P_2\downarrow^{must} \Rightarrow (P_1 \leq_R P_2 \Leftrightarrow P_1 \geq_U P_2)$
- $P_1\downarrow^{must} \Rightarrow (P_1 \leq_L P_2 \Leftrightarrow P_1 \leq_R P_2)$

Thus the relational and upper orderings are equivalent when P_2 is strongly normalising. In the

following, we will restrict our attention to the relational ordering.

Now consider the program P + Q; a correct implementation of this program would be to repeatedly transform the multiset argument using Q-steps until no further transformation occurs followed by P-steps until no transformation is possible. This latter process is a form of iterated sequential composition. The following theorem characterises the conditions under which a single sequential composition is sufficient to correctly implement the parallel composition.

Theorem 5.

$(\forall M.\ (\Phi(Q,M) \text{ and } (P,M) \rightarrow^* N) \Rightarrow \Phi(Q,N)) \Rightarrow P \circ Q \leq_R P + Q$

Proof

The divergence case follows by Proposition 5. Let us now assume: $(P \circ Q,M) \rightarrow^* N$. We have:

$\exists N'.\ (Q,M) \rightarrow^* N' \ \&\ (P,N') \rightarrow^* N$	from Proposition 3
$\Phi(Q,N')$ and $\Phi(P,N)$	from Proposition 6
$\Phi(Q,N)$	from the hypothesis
$\Phi(P,N)$ and $\Phi(Q,N)$ entails $\Phi(P+Q,N)$	from Definition 7 and the semantics of +
$(P + Q,M) \rightarrow^* (\underline{P} + \underline{Q},N)$	from Proposition 4
$\Phi(P+Q,N)$ entails $\Phi(\underline{P} + \underline{Q},N)$ and $(\underline{P} + \underline{Q},N) \rightarrow^* N$	from definitions of Φ and $\underline{=}$

Thus $(P + Q,M) \rightarrow^* N$

[]

Definition 9: $P_1 \equiv_R P_2 \Leftrightarrow P_1 \leq_R P_2 \ \&\ P_2 \leq_R P_1$

Definition 10: (P,M) is confluent, Con(P,M), iff

$$(P,M) \rightarrow^* (P_1,M_1) \ \&\ (P,M) \rightarrow^* (P_2,M_2) \Rightarrow$$
$$\exists (P_3,M_3).\ (P_1,M_1) \rightarrow^* (P_3,M_3) \ \&\ (P_2,M_2) \rightarrow^* (P_3,M_3)$$

As usual, we write Con(P) for $\forall M.\text{Con}(P,M)$.

[]

The next theorem gives sufficient conditions for sequential composition and parallel composition to be equivalent.

Theorem 6.

$(P \circ Q \leq_R P + Q \text{ and } \text{Con}(P + Q) \text{ and } (P + Q)\downarrow^{must}) \Rightarrow P + Q \equiv_R P \circ Q$

Proof

$(P \circ Q \leq_R P + Q$ and $(P + Q)\downarrow^{must}) \Rightarrow (P \circ Q)\downarrow^{must}$ by definition of $\leq_R$

$(P \circ Q \leq_R P + Q$ and $Con(P + Q)$ and $(P \circ Q)\downarrow^{must}) \Rightarrow$

$(\forall M,N.\ (P \circ Q,M) \rightarrow^* N \Leftrightarrow (P + Q,M) \rightarrow^* N)$ by definition of $\leq_R$

Indeed there is a unique N, for each M, to which both programs converge.

$((\forall M,N.\ (P + Q,M) \rightarrow^* N \Rightarrow (P \circ Q,M) \rightarrow^* N)$ and $(P + Q)\downarrow^{must}) \Rightarrow$

$P + Q \leq_R P \circ Q$ by definition of $\leq_R$

[]

So, if P + Q is confluent and strongly normalising then $P + Q \equiv_R P \circ Q$ and thus, in some contexts, we can replace sequential composition by combination. The next proposition identifies when this is safe.

Proposition 8.

$\equiv_R$ is a congruence with respect to sequential composition, $\circ$, that is:

(i) $P_1 \equiv_R P_2 \Rightarrow P \circ P_1 \equiv_R P \circ P_2$

(ii) $P_1 \equiv_R P_2 \Rightarrow P_1 \circ P \equiv_R P_2 \circ P$

[]

However, $\equiv_R$ is not a congruence with respect to +. The following is a counterexample. Consider:

P_1: $n,m \rightarrow n + m$

P_2: $n \rightarrow n - 1\ ,\ 1 \Leftarrow n > 1$

P_3: $n \rightarrow n + 1 \Leftarrow n < 10$

Then $P_1 \circ P_2 \equiv_R P_1$ but $(P_1 \circ P_2) + P_3 \not\equiv_R P_1 \circ P_3$; the latter equivalence fails because the left hand side can diverge but the right hand side cannot.

We close this section by stating a number of other useful laws relating $\circ$ and +:

Properties 1.

1. + is associative and commutative
2. $\circ$ is associative
3. $\circ$ is monotone
4. (False,A) is an identity for + and $\circ$
5. $(P_1 + P_3) \circ P_2 \leq_R (P_1 \circ P_2) + P_3$
6. $(P_1 + \underline{\underline{P}}_3) \circ (P_2 + P_3) \leq_R (P_1 \circ P_2) + P_3$
7. $(P \leq_R P + P)$ and $(P \equiv_R \underline{\underline{P}} \Rightarrow P \equiv_R P + P)$

8. $(P_1 + P_2) \circ (Q_1 + Q_2) \leq_R (P_1 \circ Q_1) + (P_2 \circ Q_2)$
9. $P_1 \circ (P_2 + P_3) \leq_R (P_1 \circ P_2) + (P_1 \circ P_3)$

Proof

We just prove 7 part (i) which illustrates the use of active contexts:

First, we have :

$\forall M.(P,M)\uparrow \Rightarrow (P + P,M)\uparrow$ by definition of $\uparrow$ and the semantics

Now suppose that:

$(P,M) \rightarrow^* N$

then there is a sequence $\{(P_i,M_i) \mid i \in \{1..n\}\}$ such that $(P_1,M_1) = (P,M)$ and $(P_n,M_n) = (\underline{P},N)$ and $(P_j,M_j) \rightarrow (P_{j+1},M_{j+1})$.

We prove that:

$(P_j,M_j) \rightarrow (P_{j+1},M_{j+1}) \Rightarrow (P_j + P_j,M_j) \rightarrow^* (P_{j+1} + P_{j+1},M_{j+1})$

from which it follows that $(P + P,M) \rightarrow^* N$.

Given $(P_j,M_j) \rightarrow (P_{j+1},M_{j+1})$, by Theorem 1, there are two cases:

Case 1: $(P_j = P_{j+1})$

$(P_j + P_j,M_j) \rightarrow (P_{j+1} + P_{j+1},M_{j+1})$ follows directly from the semantics.

Case 2: $(P_j \rightsquigarrow P_{j+1}$ and $M_j = M_{j+1})$

Notice that P_j + [] and [] + P_j are both active contexts, thus by Theorem 4 twice we have:

$(P_j + P_j,M) \rightarrow (P_j + P_{j+1},M) \rightarrow (P_{j+1} + P_{j+1},M)$

[]

4. Confluence and Termination

A Gamma program is essentially a (conditional) rewriting system; unfortunately, we cannot make use of much of the powerful theory that has been developed for such systems because the objects that we are rewriting, bags or multisets, do not have any significant structure in the sense that terms have. There are, however, some general results for Abstract Reduction Systems [11] which are useful and we review these in this section.

We recall the definition of the predicate Con in the previous section; the formulation of Con in Definition 10 states that a program is confluent if the associated rewriting relation, $\rightarrow^*$, satisfies the *diamond property*. Any rewriting relation which satisfies the diamond property is confluent or Church-Rosser (CR); in the following, it will sometimes be the case that $\rightarrow$ satisfies the diamond property, the fact that $\rightarrow^*$ satisfies the diamond property is then a simple consequence which can be proved by a diagram chase. The diamond property is quite strong and will not hold in general. The relation $\rightarrow$ is *weakly Church Rosser* (WCR) if:

$$((P,M) \rightarrow (P_1,M_1) \wedge (P,M) \rightarrow (P_2,M_2))$$
$$\Rightarrow \exists (P_3,M_3).\ ((P_1,M_1) \rightarrow^* (P_3,M_3) \wedge (P_2,M_2) \rightarrow^* (P_3,M_3))$$

The relation $\rightarrow$ is *strongly normalising* (SN) if all $\rightarrow$-rewrite sequences terminate. Given these two properties, we can still show confluence by appealing to *Newman's Lemma*:

$$WCR \wedge SN \Rightarrow CR$$

Termination can be shown in one of two ways: either by a cardinality argument, if the action strictly reduces the cardinality of the multisets involved, or by establishing a well-founded multiset ordering and showing that the rule reduces the sets with respect to this ordering. Since most of the Gamma programs that we will be considering are SN, this second approach to demonstrating confluence turns out to be very useful. A most useful ordering for proving the termination of Gamma programs is the Dershowitz-Manna multiset ordering [8]: a well-founded multiset ordering can be derived from a well-founded ordering on the elements of the multiset.

We now consider how these properties interact with the combinators that we have introduced. It is easy to verify that the sequential composition of two programs which are CR (WCR or SN) will result in a combined program which is CR (WCR or SN). Now we consider programs of the form $P_1 + P_2$. We call the rewriting relations associated with the two sub-programs $\rightarrow_1^*$ and $\rightarrow_2^*$, respectively, and we will say that they *commute* if:

$$((P,M) \rightarrow_1^* (P_1,M_1) \wedge (P,M) \rightarrow_2^* (P_2,M_2))$$
$$\Rightarrow \exists (P_3,M_3).\ ((P_1,M_1) \rightarrow_2^* (P_3,M_3) \wedge (P_2,M_2) \rightarrow_1^* (P_3,M_3))$$

Now if $\rightarrow_1^*$ and $\rightarrow_2^*$ each satisfy the diamond property and they commute, then the combined rule system is also confluent; this a consequence of the *Hindley-Rosen Lemma* which is proved by a simple diagram chase. Termination, however is not necessarily preserved - we require a stronger condition than commutativity. Here we present, a rather strong, sufficient condition; in order to weaken this condition we have to consider the particular programs. If we can show that $\rightarrow_2$-rewrites cannot create $\rightarrow_1$-redexes and if the rewrites commute, then its easy to show (by a diagram chase) that the $\rightarrow_2$-rewrites can be postponed; i.e. any rewriting sequence is equivalent to one in which all of the $\rightarrow_1$-rewrites are done first. In this situation, termination of the combined rule system follows from the separate termination of the two programs. A generalisation of this situation leads to the concept of a pipelining program, which is the subject of Section 6.

5. Definition and Properties of Tropes

The discussion of termination above highlights some of the shortcomings of the calculus that we have developed. Often we are forced to consider the particular form of the rewrites explicitly in order to make any headway. In this section we demonstrate that there are a small number of program forms from which any other program can be constructed. We have identified six of these

schemes that we call tropes (for Transmuter, Reducer, Optimiser, Permuter, Expander and Selector) that turn out to be very useful in practice. For example most of the examples in [4] can be rephrased in a straightforward way as combinations of tropes. These tropes are defined as follows:

$\mathcal{T}(C,f) \equiv x \rightarrow f(x) \Leftarrow C(x)$

$\mathcal{R}(C,F) \equiv x,y \rightarrow f(x,y) \Leftarrow C(x,y)$

$\mathcal{O}(<,f,T) \equiv x,y \rightarrow (f(y.1),x.2),y \Leftarrow f(y.1) < x.1$ and $T(x.2,y.2)$ where $<$ is a well-founded partial order

$\mathcal{P}(<_1,<_2) \equiv x,y \rightarrow (x.1,y.2),(y.1,x.2) \Leftarrow x.1 <_1 y.1$ and $x.2 <_2 y.2$ where $<_1$ and $<_2$ are both partial orders

$\mathcal{E}(C,f_1,f_2) \equiv x \rightarrow f_1(x), f_2(x) \Leftarrow C(x)$

$\mathcal{S}_{i,j}(C) \equiv x_1,\ldots,x_i \rightarrow x_j,\ldots,x_i \Leftarrow C(x_1,\ldots,x_i)$ where $1 < j \leq i + 1$

In fact it can be shown that any Gamma program can be expressed as a combination of transmuters, reducers, expanders and selectors. The proof of this involves two steps: firstly, any unary or binary program can be expressed; secondly, one has to prove that any n-ary program is expressible in terms of (n-1)-ary programs. This second part amounts to showing how part of the implementation of Gamma (the tuple handling mechanism) is expressible in Gamma itself.

Let us just come back to the examples presented in the introduction to illustrate the tropes. The programs *maxset4*, *prime_numbers*, *sort* and *maj_elem* can be expressed in the following way:

maxset4 = $\mathcal{S}_{2,2}(C)$ where $C(x,y) = x \leq y$
prime_numbers = $\mathcal{S}_{2,2}(C)$ where $C(x,y)$ = multiple (x,y)
sort = $\mathcal{P}(>,<)$ where $>$ and $<$ represent the usual orderings on integers
maj_elem = $\mathcal{S}_{2,3}(C) \circ \mathcal{S}_{2,2}$(True) where $C(x,y) = x \neq y$

Since it involves ternary reaction conditions *connected* has to be slightly rephrased to be expressed in terms of tropes:

connected' = singleton $\circ$ $(P_1 + P_2 + P_3)$ where
P_1: (v,nil), (m,n) $\rightarrow$ (v, n) $\Leftarrow$ vertices(v) and $m \in v$
P_2: (v, n), (w, m) $\rightarrow$ (v + w, m) $\Leftarrow$ vertices(v) and vertices(w) and $n \in w$
P_3: (v, n) $\rightarrow$ (v, nil) $\Leftarrow$ vertices(v) and $n \in v$

The multiset contains pairs (v,n), where v is a set of vertices and n is a vertex accessible from a

vertex in v (or the dummy value nil), and pairs (m,n) representing an edge from vertex m to vertex n. The initial multiset is {({v},nil)| v is a vertex of the graph} + {(m,n)| (m,n) is an edge of the graph}. Now *connected'* can be expressed in terms of tropes:

$$\text{connected'} = \text{singleton} \circ (\mathfrak{R}(C_1,A_1) + \mathfrak{R}(C_2,A_2) + \mathfrak{T}(C_3,A_3)) \quad \text{where}$$

$C_1((v,nil), (m,n)) = \text{vertices}(v)$ and $m \in v$

$A_1((v,nil), (m,n)) = (v, n)$

$C_2((v, n), (w, m)) = \text{vertices}(v)$ and $\text{vertices}(w)$ and $n \in w$

$A_2((v, n), (w, m)) = (v + w, m)$

$C_3((v, n)) = \text{vertices}(v)$ and $n \in v$

$A_3((v, n)) = (v, nil)$

The following solution to the prime factorization problem was presented in [4]:

$$\text{factorization}(n) = ((P_2 + P_3) \circ P_1)(\text{prime\_numbers}(n) \bullet \{(n,0)\}) \quad \text{where}$$

P_1: $(n, m, k) \rightarrow (n, m/n, k+1) \Leftarrow \text{multiple}(m, n)$

P_2: $(n, m, k) \rightarrow (n, m, k-1), n \Leftarrow k > 0$

P_3: $(n, m, k) \rightarrow \Leftarrow k = 0$

The notation $P \bullet \{(n,0)\}$ represents the multiset of triples $\{(p,n,0) \mid p \in P\}$. The program factorization returns the decomposition of an integer in terms of its prime factors. For example if $n = 2^3 * 3 * 11^2$, then factorization (n) = {2, 2, 2, 3, 11, 11}.
This definition can be rephrased in terms of tropes:

$P_1 = \mathfrak{T}(C_1,f_1)$ where $C_1((n, m, k)) = \text{multiple}(m, n)$, $f_1((n, m, k)) = (n, m/n, k+1)$

$P_2 = \mathfrak{E}(C_2,f_2,f'_2)$ where $C_2((n, m, k)) = k > 0$, $f_2((n, m, k)) = (n, m, k-1)$, $f'_2((n, m, k)) = n$

$P_3 = \mathfrak{S}_{1,2}(C_3)$ where $C_3((n, m, k)) = (k = 0)$

Since tropes embody very common patterns of Gamma programs it is important that they satisfy useful properties.

Properties 2.

1. $(\neg C(x) \;\&\; C'(x) \Rightarrow \neg C(f'(x))) \Rightarrow \mathfrak{T}(C',f') + \mathfrak{E}(C,f_1,f_2) \geq_R \mathfrak{T}(C',f') \circ \mathfrak{E}(C,f_1,f_2)$

2. $(\neg C'(x)\ \&\ C(x) \Rightarrow \neg C'(f_1(x))\ \&\ \neg C'(f_2(x))) \Rightarrow$

$\mathfrak{T}(C',f') + \mathfrak{E}(C,f_1,f_2) \geq_R \mathfrak{E}(C,f_1,f_2) \circ \mathfrak{T}(C',f')$

3. $\forall P.\ \mathfrak{S}_{i,j}(C) + P \geq_R \mathfrak{S}_{i,j}(C) \circ P$

Proof

Properties 2.1 and 2.2 are applications of Theorem 5 and Property 2.3 is a consequence of Theorem 5 and Proposition 7.

[]

Let us come back to the *factorization* example above to illustrate these properties. Since P_3 is a selector, we have:

$(P_2 + P_3) \circ P_1$	$\geq_R (P_3 \circ P_2) \circ P_1$	from Property 2.3 and Proposition 8
	$\geq_R P_3 \circ (P_2 \circ P_1)$	from Property 1.2
$P_2 + P_1$	$\geq_R P_2 \circ P_1$	from Property 2.2

The techniques described in section 4 can be used to prove $Con(P_2 + P_1)$ and $(P_2 + P_1)\downarrow^{must}$ and Theorem 6 allows us to conclude:

$$P_2 + P_1 \equiv_R P_2 \circ P_1$$

In the remainder of this section, we consider sufficient conditions for confluence and termination for each of the tropes.

5.1 Transmuter

$\mathfrak{T}(C,f) \equiv x \rightarrow f(x) \Leftarrow C(x)$

Termination

We require a well-founded ordering, $<$, on the multiset such that:

$$C(x) \Rightarrow f(x) < x$$

Confluence

There are no critical pairs and the reduction rule trivially satisfies the diamond property.

5.2 Reducer

$\mathfrak{R}(C,F) \equiv x,y \rightarrow f(x,y) \Leftarrow C(x,y)$

Termination

Termination follows trivially from cardinality considerations.

Confluence

If f is associative and commutative and C is symmetric and transitive, the rule satisfies the diamond property.

5.3 Optimiser

$O(<,f,T) \equiv x,y \rightarrow (f(y.1),x.2),y \Leftarrow f(y.1) < x.1$ and $T(x.2,y.2)$ where $<$ is a well-founded partial order.

Termination

Define the pair-ordering, $<_p$, by:

$$x <_p y \equiv (x.1 < y.1) \wedge (x.2 = y.2)$$

This ordering is well-founded because $<$ is. The rule is decreasing with respect to the Dershowitz-Manna multiset ordering induced by $<_p$ and thus strongly normalising.

Confluence

Consider the critical pair ((a,b),(c,d)). We consider three cases:

1. $b \equiv d$: then if we reduce (a,b) first we produce the multiset $(M - \{a\}) + \{(f(b.1),a.2)\}$ but then (c,b) can react to produce $(M - \{a,c\}) + \{(f(b.1),a.2),(f(b.1),c.2)\}$ and similarly if we reduce (c,d) first.

2. $a \equiv d$: then if we reduce (a,b) first we produce the multiset $(M - \{a\}) + \{(f(b.1),a.2)\}$ but then, providing f is montonic, we have $f \circ f(b.1) < f(a.1) < c.1$ and so we can rewrite using c to produce the multiset $(M - \{a,c\}) + \{(f(b.1),a.2),(f \circ f(b.1),c.2)\}$. If, on the other hand we reduce (c,a) first we get $(M - \{c\}) + \{(f(a.1),c.2)\}$, we can then rewrite (a,b) (since a is preserved by the previous rewrite) to get $(M - \{a,c\}) + \{(f(b.1),a.2),(f(a.1),c.2)\}$, and finally we can rewrite the new pair to get $(M - \{a,c\}) + \{(f(b.1),a.2),(f \circ f(b.1),c.2)\}$ if f is monotonic.

3. $a \equiv c$: then if we reduce (a,b) first we produce the multiset $(M - \{a\}) + \{(f(b.1),a.2)\}$ and, on the other hand, if we reduce (a,d) first we produce the multiset $(M - \{a\}) + \{(f(d.1),a.2)\}$. Sufficient conditions for confluence in this case are:

   (i) f is a constant function

   (ii) $<$ is a linear order, in which case the diamond degenerates to a triangle

In both cases 2 and 3 we have only establishes WCR but, together with the termination, this gives CR. In summary, sufficient conditions for confluence are that f is monotonic and that $<$ is linear (or f is constant). These arguments trivially extend to the case where we have a critical tuple (rather than pair).

5.4 Permuter

$\mathcal{P}(<_1,<_2) \equiv x,y \rightarrow (x.1,y.2),(y.1,x.2) \Leftarrow x.1 <_1 y.1$ and $x.2 <_2 y.2$ where $<_1$ and $<_2$ are both partial orders

Termination

We define the ordering $<_p$ by:

$$x <_p y \equiv (x.1 \leq_1 y.1) \wedge (x.2 \leq_2 y.2)$$

This is clearly well-founded if the sets involved are finite (as they must be). Notice that in the above rule:

$$(x.1,y.2) <_p y \text{ and } (y.1,x.2) <_p y$$

because of the reaction condition.

Confluence

Again, there are three cases for the critical pair ((a,b),(c,d)). All of the cases are similar and require that both orders are linear in order to demonstrate WCR. We consider the case (b≡d): then we can either get the multiset $(M - \{a,b\}) + \{(a.1,b.2),(b.1,a.2)\}$ $(= M_1)$ or $(M - \{c,b\}) + \{(c.1,b.2),(b.1,c.2)\}$ $(= M_2)$ and there are two cases to consider:

(i) $c.1 <_1 a.1$:

Considering M_1 above, c can react with (a.1,b.2) to produce the multiset $(M - \{a,b,c\}) + \{(a.1,c.2),\{c.1,b.2),(b.1,a.2)\}$ $(= M_3)$ and we consider two subcases:

a. $c.2 <_2 a.2$: then (a.1,c.2) and (b.1,a.2) can react to produce a and (b.1,c.2) giving M_2.

b. $c.2 >_2 a.2$: then in M_2, a can react with (b.1,c.2) to give (a.1,c.2) and (b.1,a.2) which results in M_3.

(ii) $c.1 >_1 a.1$: the argument is similar to (i).

5.5 Expander

$\mathcal{E}(C,f_1,f_2) \equiv x \rightarrow f_1(x), f_2(x) \Leftarrow C(x)$

Termination

Define S_C to be the subset of S whose elements satisfy the predicate C. Suppose that we have a well-founded ordering, <, on S_C. We say that a function, f, is *reductive* on S_C if for all $x \in S_C$, either $f(x) \notin S_C$ or $f(x) < x$. The condition for termination of an expander is that the two functions g and h are reductive on S_C.

Confluence

Since the rule has a single variable on the left hand side, there are no overlapping redexes and

confluence is trivial.

5.6 Selector

$\mathbf{S}_{i,j}(C) \equiv x_1,\ldots,x_i \rightarrow x_j,\ldots,x_i \Leftarrow C(x_i,\ldots,x_i)$ where $1 < j \leq i + 1$

Termination

Termination of this rule follows trivially by cardinality considerations.

Confluence

Given the generality of the rule, it is difficult to give conditions for confluence but, restricting ourselves to rules of the form:

$$x,y \rightarrow y \Leftarrow C(x,y)$$

A sufficient condition is that C is antisymmetric and transitive.

6. Pipelining Interfaces

The style of programming that we have proposed in the preceding sections results in programs which consist of basic programs, the tropes, combined using the $\circ$ and + combinators. Consideration of examples shows that such programs make heavy use of sequential composition. The purpose of this section is to show how the calculus of Section 3 can be used to transform a program constructed from sequential composition into a pipeline program: one in which the sequence of tasks is connected in such a way that the output of one task feeds piecemeal into the input of the next. The outcome of this transformation is that sequential composition, $\circ$, is "replaced" by parallel combination, +.

6.1 An Example

We start with a motivating example, based on various combinations of the following three tropes:

$$P_1 \equiv \mathfrak{E}((\lambda n.n > 1),(\lambda n.n-1),(\lambda n.n-2))$$
$$P_2 \equiv \mathfrak{T}((\lambda n.n=0),(\lambda n.1))$$
$$P_3 \equiv \mathfrak{R}((\lambda nm.True),+)$$

Notice that P_1 is terminating because both functions are reductive with respect to the usual ordering on positive integers (which is well-founded) and it is trivially confluent, P_2 terminates because the number of 0s is reduced by each rewrite and is trivially confluent and P_3 trivially terminates and is confluent because + is associative and commutative.

Consider first the straightforward sequential composition of these tropes:

$$P_3 \circ P_2 \circ P_1$$

If we apply this to the multiset {n}, it will terminate with a singleton multiset containing the n-th fibonnaci number. This composition effectively builds the recursion tree and then collapses it with the reducer. P_2 acts as an interface between the two processes. The first transformation is motivated by the observation that P_1 and P_2 can be performed in parallel since they do not "interfere" with one another.

Proposition 9.

For the tropes P_1, P_2 and P_3 defined above:

$$P_3 \circ (P_2 + P_1) \equiv_R P_3 \circ P_2 \circ P_1$$

Proof

Since P_2 does not produce any elements which may be rewritten by P_1, the premise of Theorem 5 holds and thus:

$$P_2 \circ P_1 \leq_R P_2 + P_1$$

$P_2 + P_1$ is confluent because P_1 and P_2 commute and are both confluent independently; consequently, by the Hindley-Rosen Lemma, their union is confluent. Termination of $P_2 + P_1$ can be argued by combining the termination of P_1 and P_2 independently with the observation that there is a one-way producer/consumer relation between P_1 and P_2; as a result of the latter and commutativity, we can show that all of the P_2 reduction steps can be postponed to the end (which is effectively what happened in the sequential composition). Thus, by Theorem 6, we have:

$$P_2 \circ P_1 \equiv_R P_2 + P_1$$

and the result follows from Proposition 8(ii).

[]

It would be nice if we could replace the remaining sequential composition by a parallel combination, producing the program $P_3 + P_2 + P_1$. Unfortunately, this is no longer confluent nor is it terminating. P_3 does not commute with P_1 because $\{2,3\} \rightarrow_{(P1)} \{1,0,3\}$ and $\{2,3\} \rightarrow_{(P3)} \{5\}$ and there are no converging P_3- and P_1-sequences from these two sets. P_3 does not commute with P_2 because $\{0,1\} \rightarrow_{(P2)} \{1,1\}$ and $\{0,1\} \rightarrow_{(P3)} \{1\}$ and again the converging sequences do not exist. We lose termination because of the possibility of cycling between rules P_1 and P_3, building ever bigger sets. The producer/consumer relationship which is implicit in the sequential composition has been lost by the arbitrary joining of the rule systems.

We can recover a concurrent producer-consumer relationship by placing an interface "between" P_3 and $(P_2 + P_1)$ which prevents interference and "passes" data from $(P_2 + P_1)$ to P_3. We proceed

by developing a general transformation scheme for pipelining before returning to this running example.

6.2 The Pipelining Transformation

We consider the problem of constructing a pipeline between programs Γ_1 and Γ_2 which correctly implements their sequential composition, $\Gamma_1 \circ \Gamma_2$. The basic idea is to identify elements of Γ_2's multiset which are *stable*, i.e. cannot partake in any further reactions in Γ_2, and "pass" these to Γ_1.

Definition 11.

An element, e, is *stable* for a program P iff

$\forall M.$ (i) $(P,M) \rightarrow^* M' \Rightarrow (P,M+\{e\}) \rightarrow^* M'+\{e\}$

(ii) $(P,M+\{e\}) \rightarrow^* (P',M') \Rightarrow (P,M) \rightarrow^* (P',M'')$ where $M' = M''+\{e\}$[4]

[]

To enable Γ_1 to "consume" the identified stable elements of Γ_2 concurrently, we must ensure that Γ_1 does not interfere with the unstable elements. This is achieved by a process of *tagging* the stable elements and by modifying Γ_1 so that it can only operate on tagged data. Let Γ_1' be the modification of Γ_1 to operate over tagged data. We do not give a formal definition of this operation, but just specify that:

- $\Gamma_1 \geq_R$ decode $\circ\, \Gamma_1' \circ$ code
  where
  code: $n \rightarrow tag(n) \Leftarrow untagged(n)$
  and
  decode : $tag(n) \rightarrow n$
- Any non-tagged element is stable for Γ_1'

Thus:

$\Gamma_1 \circ \Gamma_2$

is refined to:

decode $\circ\, \Gamma_1' \circ$ code $\circ\, \Gamma_2$

It is easy to verify the following properties of code and decode:

M contains no tagged elements $\Rightarrow$ decode(code(M)) = M

code(decode(M)) = code(M)

[4] Notice that (ii) together with Theorem 3 and Proposition 7 gives:
$(P,M+\{e\}) \rightarrow^* N \Rightarrow (P,M) \rightarrow^* N'$ where $N'+\{e\} = N$

A specification for an interface is given by the following definition:

Definition 12: A program, int, is an interface for the program Γ, if its only action is to tag elements:

$$\text{int} : n \rightarrow \text{tag}(n) \Leftarrow C(n) \text{ and } \text{untagged}(n)$$

and

$$\text{decode} \circ (\text{int} + \Gamma) \leq_R \Gamma$$

[]

There are many programs which satisfy this specification; the identity program is the weakest and the program which tags all stable elements is the strongest - any program which tags some of the stable elements will do. If we have such an interface for Γ_2 then:

Proposition 10.

If Γ_1 is simple and coded elements are stable for Γ_2[5] :

$$\Gamma_1 \circ \Gamma_2 \quad \geq_R \quad \text{decode} \circ (\text{code} + \lceil \underline{\Gamma_1} \rceil) \circ (\lceil \Gamma_1 \rceil + \text{int} + \Gamma_2)$$

Proof

$$\begin{aligned}
\Gamma_1 \circ \Gamma_2 \quad &\geq_R \quad \text{decode} \circ \lceil \Gamma_1 \rceil \circ \text{code} \circ \Gamma_2 \\
&\geq_R \quad \text{decode} \circ \lceil \Gamma_1 \rceil \circ \text{code} \circ \text{decode} \circ (\text{int} + \Gamma_2) \\
&\geq_R \quad \text{decode} \circ \lceil \Gamma_1 \rceil \circ \text{code} \circ (\text{int} + \Gamma_2) \\
&\qquad\qquad \text{by Property of code and decode} \\
&\geq_R \quad \text{decode} \circ (\lceil \Gamma_1 \rceil + \text{code} \circ (\text{int} + \Gamma_2)) \\
&\qquad\qquad \text{since by the definition of } \lceil \Gamma_1 \rceil \text{ and the} \\
&\qquad\qquad \text{premise, } \lceil \Gamma_1 \rceil \text{ and } \Gamma_2 \text{ cannot interfere} \\
&\geq_R \quad \text{decode} \circ (\text{code} + \lceil \underline{\Gamma_1} \rceil) \circ (\lceil \Gamma_1 \rceil + \text{int} + \Gamma_2) \\
&\qquad\qquad \text{by Property 1.6}
\end{aligned}$$

[]

Furthermore, we have:

Proposition 11.

If $\Phi(\lceil \Gamma_1 \rceil + \text{int} + \Gamma_2, M) \Rightarrow \Phi(\text{code} + \lceil \underline{\Gamma_1} \rceil, M)$

then:

$$\text{decode} \circ (\lceil \Gamma_1 \rceil + \text{int} + \Gamma_2) \leq_R \text{decode} \circ (\text{code} + \lceil \underline{\Gamma_1} \rceil) \circ (\lceil \Gamma_1 \rceil + \text{int} + \Gamma_2)$$

Proof

We consider two cases:

[5] This condition is satisfied by our examples; however if the condition is not satisfied it can be enforced by an appropriate tagging.

i. decode $\circ$ (Γ_1' + int + Γ_2)$\uparrow$ $\Rightarrow$ (Γ_1' + int + Γ_2)$\uparrow$

$\Rightarrow$ (decode $\circ$ (code + $\underline{\Gamma}_1$') $\circ$ (Γ_1' + int + Γ_2))$\uparrow$

ii. Let decode $\circ$ (Γ_1' + int + Γ_2) $\rightarrow^*$ M, for some M. Then there is some multiset N such that:

(Γ_1' + int + Γ_2) $\rightarrow^*$ N and Φ((Γ_1' + int + Γ_2),N)

But then, by assumption, Φ(code +$\underline{\Gamma}_1$',N) and thus:

(decode $\circ$ (code + $\underline{\Gamma}_1$') $\circ$ (Γ_1' + int + Γ_2)) $\rightarrow^*$ M

[]

In particular, it is easy to verify that, this proposition does apply if int is complete (i.e. tags all stable elements).

6.3 Further Examples

Returning to our example the only stable elements for $P_2 + P_1$ are the 1s; since it is possible to define a complete interface, the program $P_3 \circ (P_2 + P_1)$ can be transformed to:

decode $\circ$ (P_3' + int + P_2 + P_1)

where:

P_3' : a, b $\rightarrow$ tag(x + y) $\Leftarrow$ a = tag(x) $\wedge$ b = tag(y)

int : n $\rightarrow$ tag(1) $\Leftarrow$ n = 1

To close this section we present another example of this pipelining transformation. We consider the prime factorisation problem, presented in [4] and in Section 5 of this paper, which utilises the result from number theory that any number can be written uniquely as a product of primes. The program has a number as its input[6] and produces a multiset of primes, each prime factor of the input being repeated the number of times that it is used in the factorisation.

We define the initial program as follows:

factor(n) $\equiv$ ($P_6 \circ P_5 \circ P_4 \circ P_3 \circ P_2 \circ P_1$) {(n,n)}

where

P_1: (a,b) $\rightarrow$ (a,b,0),(a-1,b) $\Leftarrow$ a $\geq$ 3

P_2: (a,b) $\rightarrow$ (a,b,0) $\Leftarrow$ a < 3

P_3: (x,a,b),(y,c,d) $\rightarrow$ (y,c,d) $\Leftarrow$ multiple(x,y)

P_4: (n_1,n_2,k) $\rightarrow$ (n_1,n_2/n_1,k+1) $\Leftarrow$ multiple(n_2,n_1)

P_5: (n_1,n_2,k) $\rightarrow$ (n_1,n_2,k-1),n_1 $\Leftarrow$ k $\geq$ 1

P_6: (n_1,n_2,k) $\rightarrow$ $\Leftarrow$ k = 0

P_1 and P_2 together produce a (multi)set of triples such that each element consists of a number less

[6] This is slightly different from the earlier program in order to give us a more significant example to apply our transformation to.

than or equal to the original input, the original input and 0; P_3 removes all of the triples which do not have a prime number as their first component; P_4 increments the third component of the triples to record the number of times that the prime first element divides the input; P_5 generates copies of the primes; and P_6 deletes redundant triples. P_4, P_5 and P_6 correspond to P_1, P_2 and P_3 of Section 5.

We make the following observations about factor:

(i) $P_3 \circ P_2 \circ P_1 \equiv_R P_3 + P_2 + P_1$

since the premise of Theorem 5 holds each rule is terminating and confluent separately and they commute, thus Theorem 6 applies.

(ii) $P_6 \circ P_5 \equiv_R P_6 + P_5$

by analogy to the first part of the proof of Proposition 9 (note that the earlier solution to this problem already used + rather than $\circ$ to combine these two programs).

We also have:

(iii) $P_4 \circ (P_3 + P_2 + P_1) \equiv_R P_4 + P_3 + P_2 + P_1$

since as P_4 does not modify the first element of any triple, the premise of Theorem 5 holds. P_4 clearly commutes with P_1 and P_2 but it also commutes with P_3; thus, since P_4 is confluent, $P_4 + P_3 + P_2 + P_1$ is confluent. P_1 and P_2 may produce P_4 redexes, P_3 deletes P_4 redexes but P_4 does not produce P_1, P_2 or P_3 redexes (although it may modify P_3 redexes); consequently P_4 redexes may be postponed and termination follows from the separate termination of P_4 and $(P_3 + P_2 + P_1)$. Thus Theorem 6 applies.

By Proposition 8(ii) twice, these results give:

$\text{factor}(n) \equiv_R ((P_6 + P_5) \circ (P_4 + P_3 + P_2 + P_1))\ \{(n,n)\}$

The final composition is a candidate for the pipelining transformation.

Unfortunately, the only interface that we can define for $(P_4 + P_3 + P_2 + P_1)$ is the trivial interface (the identity program) which leads to no improvement. However, it is possible to define an interface, int, for $P_4 \circ (P_3 + P_2 + P_1)$:

$\text{int}: (n_1,n_2,k) \rightarrow \text{tag}(n_1,n_2,k) \Leftarrow \neg\, \text{multiple}(n_2,n_1)$

It is routine to verify that:

$\text{decode} \circ (\text{int} + (P_4 \circ (P_3 + P_2 + P_1))) \leq_R (P_4 \circ (P_3 + P_2 + P_1))$

Now we can define:

$\mathcal{P}_5$': $\text{tag}(n_1,n_2,k) \rightarrow \text{tag}(n_1,n_2,k\text{-}1),\text{tag}\ n_1 \Leftarrow k \geq 1$

$\mathcal{P}_6$': $\text{tag}(n_1,n_2,k) \rightarrow \Leftarrow k = 0$

and it is easy to verify that:

$\Phi(\mathcal{P}_6' + \mathcal{P}_5' + \text{int} + (P_4 \circ (P_3 + P_2 + P_1)),M) \Rightarrow \Phi(\text{code} + \underline{\mathcal{P}_6}' + \underline{\mathcal{P}_5}',M)$

and consequently, factor(n) can be implemented by:

$\text{decode}((\mathcal{P}_6' + \mathcal{P}_5' + \text{int} + (P_4 \circ (P_3 + P_2 + P_1)))\{(n,n)\})$

A final optimisation which is possible is to omit the tagging of the second component of the right hand side of 'P_5'; such single elements can play no further part in the computation.

7. Conclusion

It is recognized that a complete theory of programming should include "a method of transforming programs to achieve a high efficiency on the machine available for their execution" [10]. This paper is a preliminary attempt to provide such a program calculus for Gamma. Our first task was to define appropriate composition operators for Gamma programs. This effort should be related to the work done for UNITY in [5, 13]. The parallel operator of UNITY (||) is very similar to our + operator. For instance the property [13]:

FP of F || G = (FP of F) & (FP of G)

corresponds to

$\Phi(F + G,M) \Leftrightarrow \Phi(F,M) \;\&\; \Phi(G,M)$

in our setting.

However our approach departs from the UNITY work in several important aspects: we describe a calculus for the transformation of Gamma programs whereas the || operator of UNITY is defined by a number of theorems in the UNITY logic. Also || is the only composition operator in UNITY whereas our main concern in this paper was to relate the sequential and the parallel compositions of Gamma.

It should be clear that the paper is a preliminary attempt at designing a calculus for Gamma programs. As stated in section 3, $\equiv_R$ is not a congruence. An interesting area for further research would be to define the largest pre-congruence included in $\leq_R$. Also other operators might be worth adding to the Gamma calculus. Let us mention for example the iterator * satisfying the following:

$$\frac{(P_2,M_2) \rightarrow^* M'_2 \quad (P_1,M'_2) \rightarrow^* M'_1}{(P_1 * P_2, M_2) \rightarrow (P_1 * P_2, M'_1)} \qquad \frac{(P_1,M_1) \rightarrow^* M_1 \quad (P_2,M_1) \rightarrow^* M_1}{(P_1 * P_2, M_1) \rightarrow M_1}$$

$(P_1 * P_2, M)$ behaves as the repeated application of P_2 and P_1 until a multiset is obtained, which is stable for P_1 and P_2. It satisfies the following properties:

$P_1 \equiv_R \underline{P}_1 \;\&\; P_2 \equiv_R \underline{P}_2 \Rightarrow P_1 * P_2 \leq_R P_1 + P_2$

and $(P_1 \circ P_2, M) \rightarrow M' \Rightarrow (P_1 * P_2, M) \rightarrow (P_1 * P_2, M')$

Gamma has also been used for describing (non terminating) reactive programs [4]. We intend to study the implications of these new operators on reactive programs and the relationship with other

works on communicating process languages.

The work of Back [1] on refining atomicity in parallel algorithms is related in spirit to our study of transformations between sequential and parallel compositions. Providing a more "axiomatic" account of our work might reveal some deeper connections. The introduction of (explicit) sequential composition into the Gamma model was motivated in part by the desire to address efficiency and architectural considerations at an abstract algorithmic level. Similar motivations are present in the recent work of Liu, Singh and Bagrodi [12], in the context of the UNITY logic.

References

[1] Back R. J. R. *A Method for Refining Atomicity in Parallel Algorithms.* In PARLE '89, volume II, Springer-Verlag, LNCS 365.

[2] Banâtre, J.-P., Coutant, A., and Le Métayer, D. *A Parallel Machine for Multiset Transformation and its Programming Style.* Future Generation Computer Systems. 4 (1988) 133-144.

[3] Banâtre, J.-P., and Le Métayer D. *The Gamma model and its discipline of programming.* Science of Computer Programming 15 (1990) 55-77.

[4] Banâtre, J.-P., and Le Métayer D. *Programming by multiset transformation.* Comm. of the ACM, to appear.

[5] Chandy K. M., and Misra J., *Parallel program design: a foundation*, Addison-Wesley, 1988.

[6] Creveuil C., *Implementation of Gamma on the Connection Machine*. Proc. of the Workshop on Research Directions in High-level Parallel Programming Languages, Springer Verlag, LNCS 574, 1992.

[7] Creveuil C., *Techniques d'analyse et de mise en œuvre de programmes Gamma.* Thèse de 3ème cycle, Université de Rennes 1, December 1991.

[8] Dershowitz N., Manna Z., *Proving termination with multiset orderings*, Communications of the ACM, 22, 8, pp. 465-476, 1979.

[9] Hillis W. D., *The connection machine*, MIT Press, Cambridge, Mass, 1985.

[10] Hoare C.A.R., foreword of [5].

[11] Klop J. W., *Term rewriting systems*, Report CS-R9073, CWI, Amsterdam, 1990.

[12] Liu Y., Singh A. K., and Bagrodia, R. L. *A Decompositional Approach to the Design of Efficient Parallel Programs.* In PARLE '92, Springer-Verlag, LNCS 605.

[13] Misra J., *A foundation of parallel programming*, in Constructive Methods in Computer Science, NATO ASI series, Vol. F55, 1989.

[14] Mussat L., *Parallel programming with bags.* Proc. of the Workshop on Research Directions in High-level Parallel Programming Languages, Springer Verlag, LNCS 574, 1992.

LISTE DES DERNIERES PUBLICATIONS INTERNES PARUES A L'IRISA

PI 667 MODELES D'EVALUATION DE LA FIABILITE DU LOGICIEL ET TECHNIQUES DE VALIDATION DE SYSTEMES DE PREDICTION : ETUDE BIBLIOGRAPHIQUE
James LEDOUX
Juillet 1992, 76 pages.

PI 668 TWO COMPLEMENTARY NOTES ON SKEWED-ASSOCIATIVE CACHES
André SEZNEC
Juillet 1992, 10 pages.

PI 669 PARALLELISATION D'UN ALGORITHME DE DETECTION DE MOUVEMENT SUR UNE ARCHITECTURE MIMD
Fabrice HEITZ, Sergui JUFRESA, Etienne MEMIN, Thierry PRIOL
Juillet 1992, 34 pages.

PI 670 UN RESEAU SYSTOLIQUE INTEGRE POUR LA CORRECTION DE FAUTES DE FRAPPE
Dominique LAVENIER
Juillet 1992, 120 pages.

PI 671 EARLY WARNING OF SLIGHT CHANGES IN SYSTEMS AND PLANTS WITH APPLICATION TO CONDITION BASED MAINTENANCE
Qinghua ZHANG, Michèle BASSEVILLE, Albert BENVENISTE
Juillet 1992, 32 pages.

PI 672 ORDRES REPRESENTABLES PAR DES TRANSLATIONS DE SEGMENTS DANS LE PLAN
Vincent BOUCHITTE, Roland JEGOU, JeanXavier RAMPON
Juillet 1992, 8 pages.

PI 673 AN EXCEPTION HANDLING MECHANISM FOR PARALLEL OBJECT-ORIENTED PROGRAMMING
Valérie ISSARNY
Août 1992, 36 pages.

PI 674 A CALCULUS OF GAMMA PROGRAMS
Chris HANKIN, Daniel LE METAYER, David SANDS
Juillet 1992, 32 pages.

ISSN 0249 - 6399